Recursion continued

Arnamoy Bhattacharyya

*Most slides inspired by Bogdan Simion and Danny Heap

Overview

1. Today + next lecture → experts in structural recursion
2. How to write solution to recursive problems
3. General recipe for recursion
    a. Input is a recursive data structure (list of lists, recursive trees etc.)

Recursion General Structure

```
def recursive_function(recursive_data):

    if condition:

        do not use recursion

    else:

        Use recursion
```

Example: Depth of a List

Define the depth of list as 1 plus the maximum depth of list 's elements if list is a list, otherwise 0

```
if isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ is not a list
    return 0
# find the bug! (then fix it...)
```

Where recursion is not necessary?

Example: Depth of a List

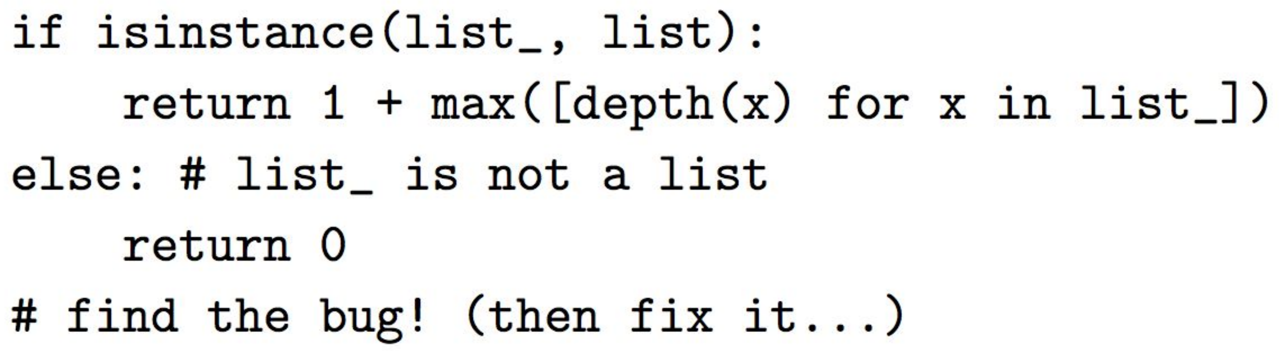
Define the depth of list as 1 plus the maximum depth of list 's elements if list is a list, otherwise 0

```
if isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ is not a list
    return 0
# find the bug! (then fix it...)
```

Where recursion is not necessary? In the Base case

Example: Depth of a List

Define the depth of list as 1 plus the maximum depth of list 's elements if list is a list, otherwise 0

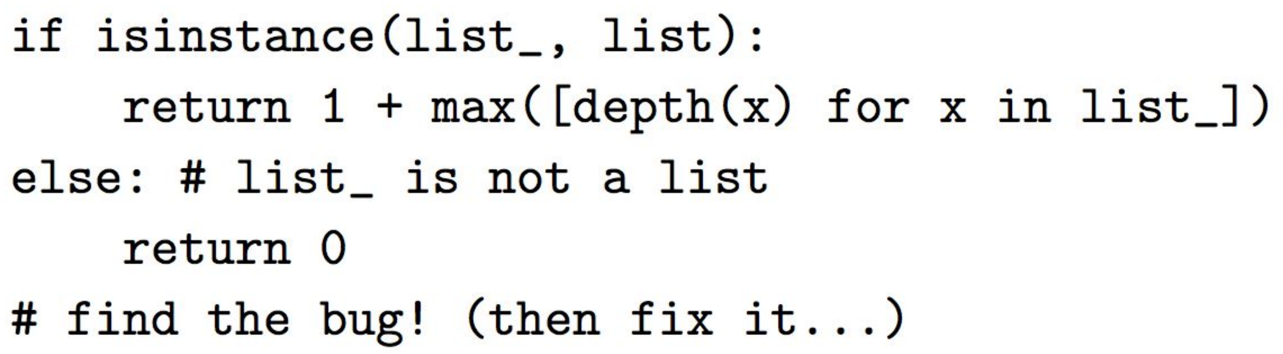


```
if isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ is not a list
    return 0
# find the bug! (then fix it...)
```

What are the simpler cases in the if-branch?

Example: Depth of a List

Define the depth of list as 1 plus the maximum depth of list 's elements if list is a list, otherwise 0



```
if isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ is not a list
    return 0
# find the bug! (then fix it...)
```

What are the simpler cases in the if-branch? Smaller and smaller lists

Example: Depth of a List

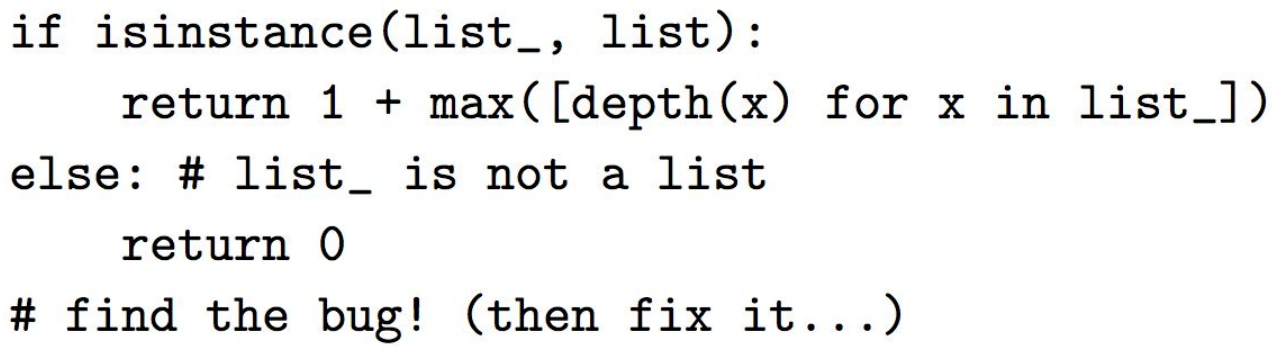
Define the depth of list as 1 plus the maximum depth of list 's elements if list is a list, otherwise 0

```
if isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ is not a list
    return 0
# find the bug! (then fix it...)
```

What is the assumption in the if-branch?

Example: Depth of a List

Define the depth of list as 1 plus the maximum depth of list 's elements if list is a list, otherwise 0



```
if isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ is not a list
    return 0
# find the bug! (then fix it...)
```

What is the assumption in the if-branch? depth() works in smaller cases

Gaining confidence: tracing recursion

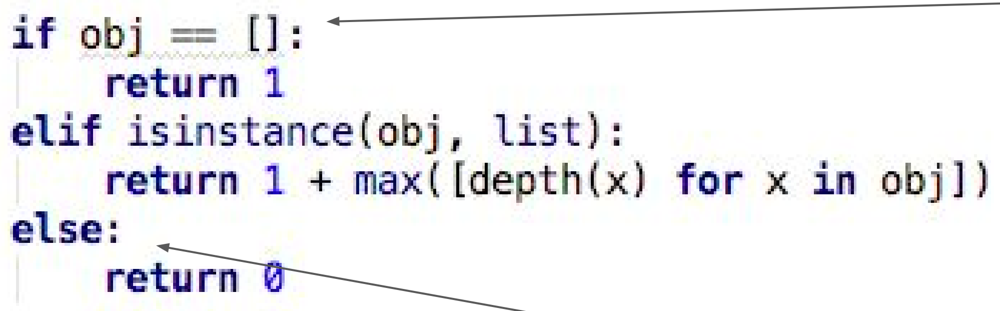
```
if isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ is not a list
    return 0
# find the bug! (then fix it...)
```

- Trace `depth([])`
- Trace `depth(17)`
- Trace `depth([3, 17, 1])`
- Trace `depth([5, [3, 17, 1], [2, 4], 6])`
- Trace `depth([14, 7, [5, [3, 17, 1], [2, 4], 6], 9])`

depth() implementation

Onto Pycharm

Base case, General Case



```
if obj == []:
    return 1
elif isinstance(obj, list):
    return 1 + max([depth(x) for x in obj])
else:
    return 0
```

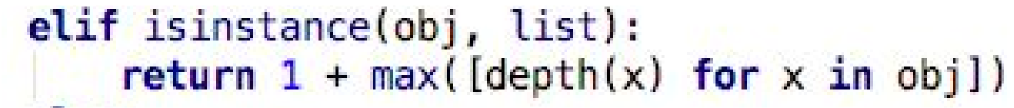
Base Case

General Case

Base Case

Base Case + General Case

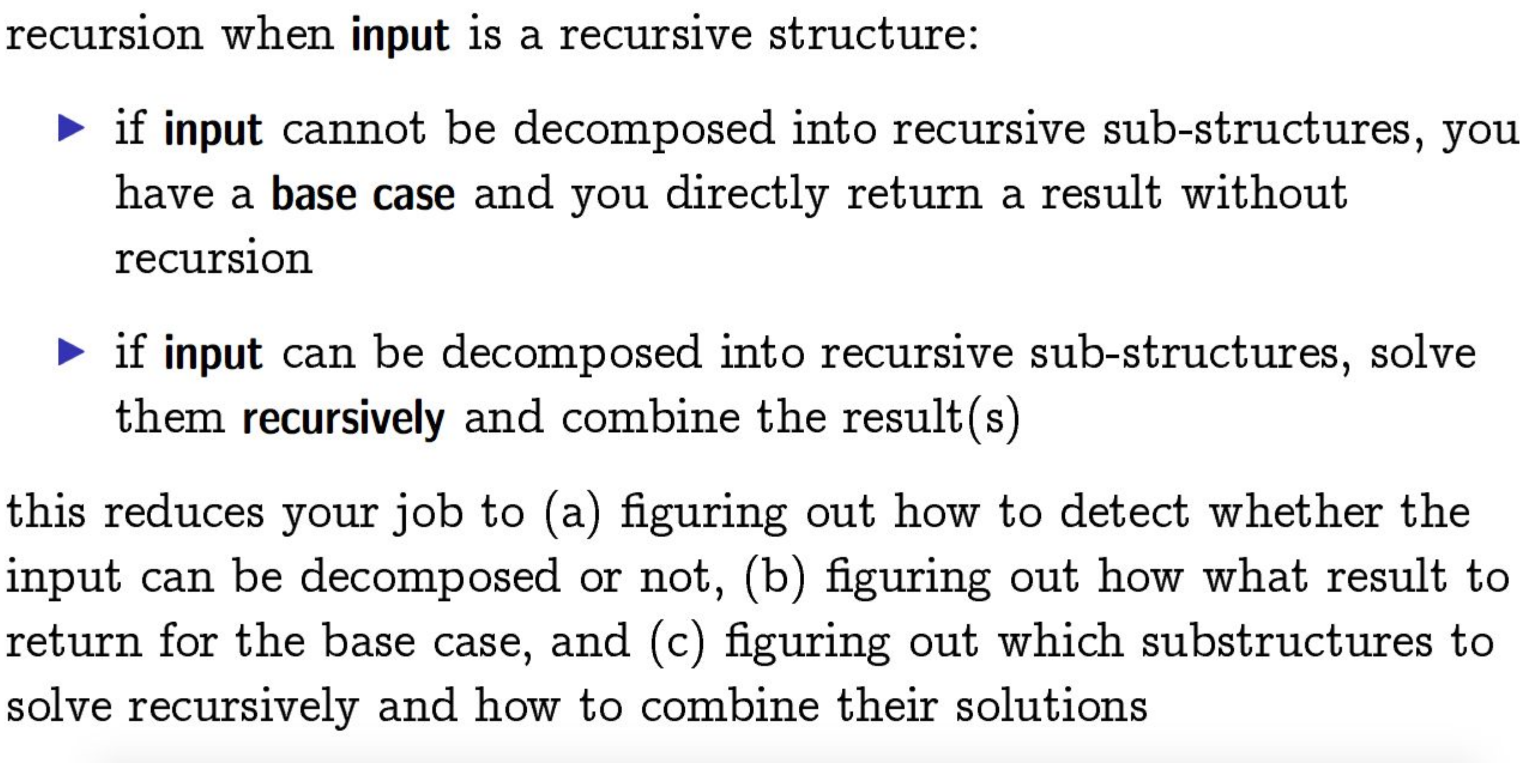
1. If you do not have a base case, recursion will run forever
2. Identify the base case where recursion will end
3. Everything else, that can be broken down into smaller problems → general case
4. Real work is how to combine the results for general case



```
elif isinstance(obj, list):
    return 1 + max([depth(x) for x in obj])
```

5. Once you have figured out how to combine both, you are golden

General Recipe for Recursion



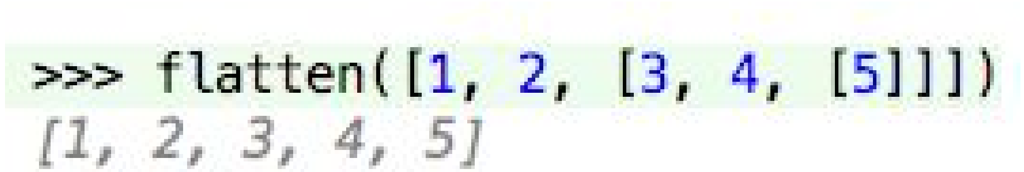
recursion when **input** is a recursive structure:

- if **input** cannot be decomposed into recursive sub-structures, you have a **base case** and you directly return a result without recursion
- if **input** can be decomposed into recursive sub-structures, solve them **recursively** and combine the result(s)

this reduces your job to (a) figuring out how to detect whether the input can be decomposed or not, (b) figuring out how what result to return for the base case, and (c) figuring out which substructures to solve recursively and how to combine their solutions

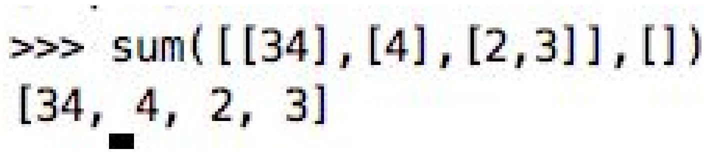
Example 2: flattening a list

Flatten a list of lists to a list of depth 1



```
>>> flatten([1, 2, [3, 4, [5]]])
[1, 2, 3, 4, 5]
```

Hint: We will use the following python trick

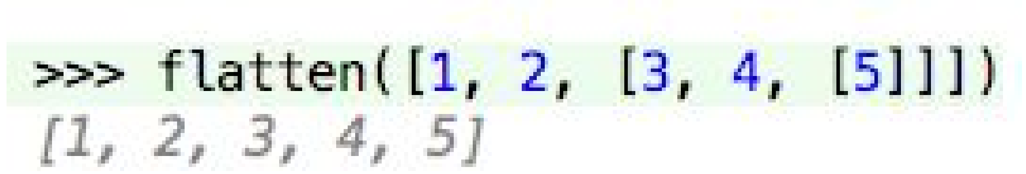


```
>>> sum([[34],[4],[2,3]],[])
[34, 4, 2, 3]
```

Example 2: flattening a list

Idea:

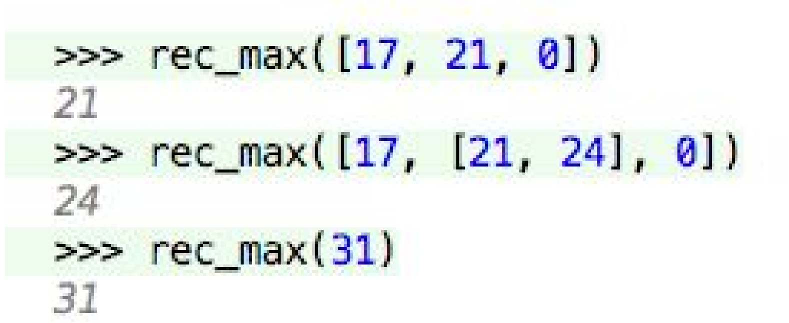
1. Keep recursing until the non-list elements of the original list
2. Make a 1d list of the non-list elements of the list
3. Use sum to flatten from the 1-d lists



```
>>> flatten([1, 2, [3, 4, [5]]])
[1, 2, 3, 4, 5]
```

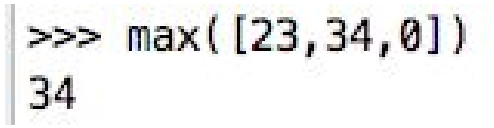
Example 3: Max from nested list

Find the max from a recursive list structure



```
>>> rec_max([17, 21, 0])
21
>>> rec_max([17, [21, 24], 0])
24
>>> rec_max(31)
31
```

Hint: We will be using the built-in max

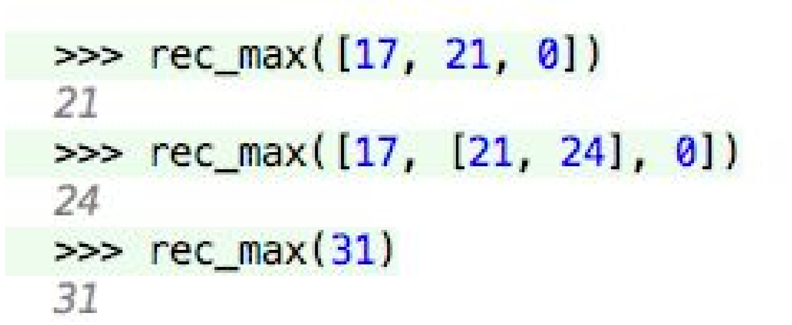


```
>>> max([23,34,0])
34
```

Example 3: Max from nested list

Idea

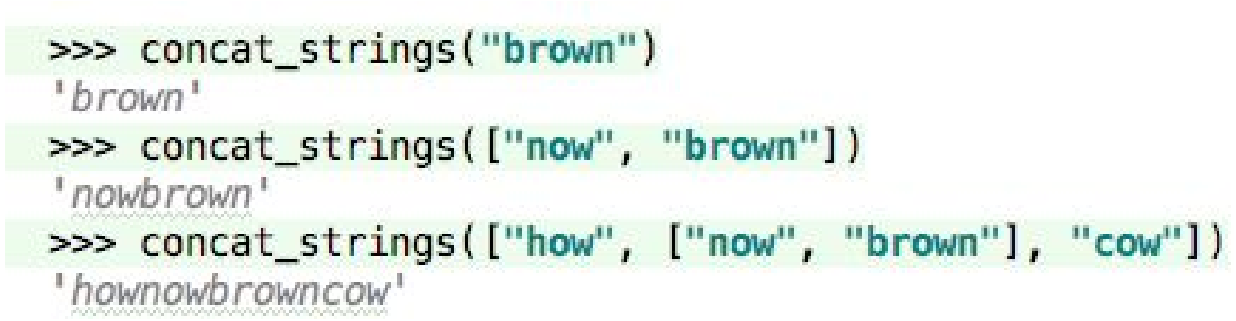
1. Keep recursing until the non-list elements of the original list
2. Return the non-list element
3. Find max from the list of non-list elements



```
>>> rec_max([17, 21, 0])
21
>>> rec_max([17, [21, 24], 0])
24
>>> rec_max(31)
31
```

Example 4: Concatenate Strings

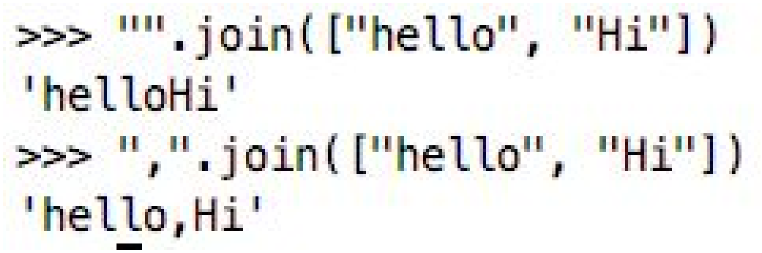
Concatenate nested list of strings



```
>>> concat_strings("brown")
'brown'
>>> concat_strings(["now", "brown"])
'nowbrown'
>>> concat_strings(["how", ["now", "brown"], "cow"])
'hownowbrowncow'
```

Hint: We will be using the str.join() method

of python



```
>>> "".join(["hello", "Hi"])
'helloHi'
>>> ",".join(["hello", "Hi"])
'hello,Hi'
```

Example 4: Concatenate Strings

Idea:

1. Keep recursing until non-list elements of the original list
2. Return the non-list element (string)
3. Use join to concatenate the list of strings